Compiler-Assisted Test Acceleration on GPUs for Embedded Software

Compiler-Assisted Test Acceleration on GPUs for Embedded Software

Anonymous Author(s)

ABSTRACT

Embedded software is found everywhere from our highly visible mobile devices to the confines of our car in the form of smart sensors. Embedded software companies are under huge pressure to produce safe applications that limit risks, and testing is absolutely critical to alleviate concerns regarding safety and user privacy. This requires using large test suites throughout the development process, increasing time-to-market and ultimately hindering competitivity.

Speeding up test execution is, therefore, of paramount importance for embedded software developers. This is traditionally achieved by running, in parallel, multiple tests on large-scale clusters of computers. However, this approach is costly in terms of infrastructure maintenance and energy consumed, and is at times inconvenient as developers have to wait for their tests to be scheduled on a shared resource.

We propose to look at exploiting GPUs (Graphics Processing Units) for running embedded software testing. GPUs are readily available in most computers and offer tremendous amounts of parallelism, making them an ideal target for embedded software testing. In this paper, we demonstrate, for the first time, how test executions of embedded C programs can be automatically performed on a GPU, without involving the end user. We take a compiler-assisted approach which automatically compiles the C program into GPU kernels for parallel execution of the input tests. Using this technique, we achieve an average speedup of 16× when compared to CPU execution of input tests across nine programs from an industry standard embedded benchmark suite.

1 INTRODUCTION

The embedded software market has grown at an extraordinary pace over the last decade. Embedded systems are ubiquitous, featuring in consumer electronics like mobile communication devices, safety critical systems such as car sensors, breaking systems, medical monitoring devices and telecommunication systems. The widespread use and close daily interaction with these systems makes safety concerns a top priority when developing and approving embedded software. Testing such software is crucial for gaining confidence on their safety, and limiting risks to users and companies.

1.1 Problem

Although individual embedded software code bases tend to be orders of magnitudes smaller than traditional applications, their test suites are typically large due to the focus on safety. As a result, the development process is hindered by the constant need for running large test suites, increasing costs and slowing down innovation. Speeding up the testing process of embedded software is critical for maintaining fast time-to-market in a competitive market. This need is exacerbated by development practices such as Test-Driven Development (TDD) that relies on short development cycles and repeated runs of tests for better quality software. Software companies in this domain are able to confirm this observation - Keysight Technologies that develop software for the telecommunication industry have a constant need to test state machines that are typically small but require a large number of tests, to achieve confidence in their correctness. They state that, "Testing the finite state machines is part of the test cycles for TDD, the shorter we can keep the time to execute the test suites, while achieving full coverage, the better. The flip side is, the more coverage we add the slower the tests become and it slows down the TDD and Continuous Integration cycles." Oxford Wave Research who specialize in audio fingerprinting software, require large test suites for fairly small code bases in their products, and state that "the time taken to run all our tests on one of our products is approximately 60 hours".

Managing limited battery power when running in-situ test suites periodically on the embedded device is also a concern [14]. Existing solutions that have focused on reducing the number of test cases to be executed are not suitable to the field of embedded software since it comes at the expense of fault finding effectiveness [11, 12], departing from the crucial goal of ensuring safety. Therefore, new approaches are necessary that can maintain the large number of test cases while accelerating the overall test execution process.

To combat the problem with time consuming test runs during development, industry is moving towards distributing test execution among multiple machines, executing them concurrently to reduce total run time. This approach, however, is costly in terms of resources, infrastructure, maintenance and energy consumed. GPUs (Graphics Processing Units) have emerged as formidable parallel accelerators that are easily available, present in virtually any modern computer from desktop workstations to laptops. We believe GPUs will find particular use in testing in-development code by allowing developers to run tests quickly on their local machine instead of waiting to access a shared resource. Rajan et al. [18] have shown that GPUs offer large potential for accelerating test suite executions. However, they manually transform programs and tests to run on the GPU. GPUs are notoriously hard to program and porting an existing application to the GPU might introduce bugs on its own, defeating the purpose.

ISSTA 2017, Santa Barbara, CA
2016. 978-x-xxxx-xxxx-x/YY/MM. . . $15.00
DOI: 10.1145/nnnnnnn.nnnnnnn

1.2 Contributions

In this paper, we show that it is possible to automatically accelerate embedded software test executions on the GPU without requiring any GPU programming knowledge. We take a compiler approach to solve this problem by automatically generating GPU code from the original unmodified C programs. We apply classical, time-proven, compiler transformations that offer higher levels of guarantees than manual porting of the original program to GPU. In addition to GPU code generation, our tool-chain automatically handles data transfers to and from the GPUs, completely relieving the programmers from writing any GPU-specific code. It is worth noting that we focus on functional testing, purely concerned with the implementation behavior, so we can safely offload the test execution on a different hardware.

We evaluate our approach using applications from the automotive and telecom domain of the EEMBC industry-standard benchmarks for embedded systems. The EEMBC benchmark suite is designed to reflect real-world embedded systems applications. It is created and maintained by an industry consortium that includes major embedded systems industry players like Samsung, Sony and Huawei. For each benchmark, we automatically produce a GPU kernel which is run in parallel across the input tests. Our results show that running the generated GPU code on an Nvidia GPU achieves a speedup of up to 53×, and an average speedup of 16× over single thread performance across benchmarks, compared to a speedup of 7× achieved with an 8 core CPU. We also offer experimental evidence that the sequential program semantics remains unchanged with our approach.

To summarize, this paper makes the following contributions:

(1) Presents an approach for automatically generating GPU code from sequential C embedded applications for accelerating testing.
(2) Improves usability by automatically launching test executions on the GPU without requiring any expert knowledge from programmers.
(3) Empirical evaluations on industry standard embedded system benchmarks for correctness and performance improvement.
(4) Analysis of benchmark features that are favorable for speedup on GPUs.

The rest of this paper is organized as follows. Sections 2 and 3 discusses background and related work. Section 4 presents our approach which includes GPU code generation and runtime. Our experimental methodology is described in Section 5. Section 6 presents and discusses the results from our experiments. Future work is presented in Section 7 while section 8 concludes this paper.

2 BACKGROUND

Generally, GPUs consist of one or more *compute units* (a.k.a *streaming multiprocessors*), which in turn contain one or more *processing elements* (a.k.a *streaming processors*). The processing elements execute groups of individual threads, referred to as work-groups or blocks depending on the GPU programming model, concurrently. The functions executed by the GPU threads are called *kernels*. GPUs have SIMD (single instruction, multiple data) architecture, in which all threads in a work-group execute the same kernel with different input data. This makes them a good fit for the execution of functional software tests, in which the tested functionality is executed multiple times with different test inputs.

A work-group is further divided into groups of threads (typically 32) called warps. Threads belonging to a warp are executed in *lock-step*, i.e all threads in a warp execute the same instruction but on different data. If there is control-flow divergence among threads in a warp, divergent instructions will be serialised, impacting performance negatively. This has implications for testing when test cases launched on the same warp take different control-flow paths. We plan to investigate methods to mitigate this effect in our future work.

The two most widely used programming models for GPU programming are CUDA [1] and OpenCL [2]. Based on C/C++ programming languages, they expose low-level hardware details, which require the programmer to explicitly express the parallelism in their algorithms. The mapping of parallelism to compute units and threads is also explicitly managed. OpenCL is our programming model of choice, since it provides cross-platform functional portability, which could potentially enable future research on testing using different accelerator architectures. Groups of threads executing the same kernel on a compute unit and sharing local memory is referred to as a work-group in OpenCL.

OpenCL is a C-like language and as a result testing C programs by translating them into an OpenCL kernel using our approach is uncomplicated. However, GPUs are specialised and not all C/C++ features are supported by the OpenCL compiler, which has implications for the scope of C applications which can be tested on the GPU. Our tool design takes this into consideration and includes methods to handle it.

3 RELATED WORK

Existing work in reducing test suite execution time is primarily focused on reducing the size of the test suite [23]. However, it has been shown that reduction in test suite sizes can result in reduced fault finding effectiveness [12],[11]. Additionally, the criteria used to reduce test suite sizes – structural coverage of the code, does not have a strong correlation with effectiveness when size of the test suite is controlled for [12]. As a result, reducing execution time by discarding tests may not be the preferred approach. Industry has looked at distributing test execution using multiple machines. Patent by Kushneryk et al. [13] uses an auxiliary test environment[1] to run test cases in parallel with the primary test environment. In this paper, we accelerate test suite execution by effectively leveraging parallel accelerators such as GPUs, a

[1]They define environment to be well-known computing systems like PCs, laptops, servers, etc.

topic that has received little attention in the software testing community.

General-purpose computing on GPUs (GPGPU) has been successfully applied in a broad range of domains [6, 16, 20]. The key challenge, in general, is the identification of opportunities to parallelize. There is growing interest in the software engineering community to use the massive performance advantage offered by GPUs. Recenty, Yu et al. explored the use of GPUs for test case generation [25]. Bardsley et al. have developed a static verfication tool, GPUVerify, for analyzing GPU kernels [7]. Li et al. [15] and Yoo et al. [24] have adapted multi-objective evolutionary algorithms for test suite optimization to execute on GPUs. They transform the *algorithm* for regression test prioritization or minimization to execute on a GPU, which is very different from the objective and methodology in our approach.

The only paper to explore the use of GPUs to accelerate test suite *execution* is [18], by Rajan et al. The key points in their approach are: The program and its logic remains unchanged. The changes to execute the tests in parallel on the GPU are only to the program interface. The program functionality is launched as a GPU kernel with each thread using a different test input data. The approach in [18], however, uses manual code transformation and is incomplete in tackling GPU limitations with respect to ease of programming, unsupported program features, and performance optimizations. The paper also lacks a detailed evaluation and analysis of when the proposed technique will be useful. In this paper, we tackle the feasibility and ease of use challenge by designing a framework that allows test cases to be *automatically* launched on the GPU without requiring any GPU programming knowledge and improving supported program features. We also show, through empirical evaluations, that the framework preserves correctness and generates significant speedup on industry standard embedded systems benchmarks. We present detailed analysis of our results and identify features in benchmarks that are suitable/unsuitable for test execution on the GPU.

GPU programming poses many challenges for the developer, both in terms of programmability and performance. The use of low-level programming models, such as CUDA and OpenCL, requires familiarity with the architecture in order to write correct parallel code, and effective optimizations in order to reach the full performance potential of the GPU. Previous research addresses these challenges by proposing high-level programming frameworks, compilers and code generation tools. For instance, [19, 22] introduce and evaluate a framework, which automatically generates low-level OpenCL code from high-level parallel primitives. The work defines the primitives, which correspond to parallel functionalities, e.g. *map* and *reduce*, as well as a functional-style programming language which is used by the programmer to express parallel algorithms. Another example is SYCL [3], which provides a high level-abstraction of OpenCL to allow programmers to write GPU code in standard C++. Purely compiler-based approaches include [8], targeting CUDA, and [10], aimed at lower-level code generation. Both of them use the polyhedral model for loop parallelization in order to transform portions of the program into parallel code to be executed on the GPU.

These existing tools and frameworks provide high-level mechanisms to both discover and express parallelism for the GPU. They are, however, not suited for our purposes since the need in our approach is *not* identification of parallelism, as that is inherent in test execution - the test cases are directly mapped to the GPU threads. The need lies in a code generation tool that will take the CPU program and transform it into an OpenCL kernel, without affecting the core program functionality, while also launching it with a different test case on each GPU thread. None of the existing tools can provide this capability. In the next sections, we describe the design and implementation of our framework that addresses these needs.

4 OUR APPROACH

To automate the testing process, we have implemented a framework, illustrated in Figure 1 with the following goals, (1) Abstract away low level GPU details, making the approach accessible to all programmers, even those unfamiliar with OpenCL, and (2) Allow automatic transformations of program features typically unsupported on the GPU. Our framework consists of two components,

(1) **CodeGen - a code generation tool.** It translates the tested C program to an OpenCL kernel, which can be executed by the GPU threads. It also generates data structures and functions, used by the runtime, to transfer test cases and results between the CPU and GPU memories. To do this, it takes the unmodified source code of the tested program and a configuration file as inputs.

(2) **Runtime - a test execution system.** The runtime executes on the CPU. It performs the following steps: (1) It reads the test cases, supplied in CSV format, and transfers them to the GPU memory. (2) It builds the OpenCL kernel, generated from the tested program, and launches it in parallel on the GPU threads. (3) When the GPU finishes execution, it transfers the testing results back to the CPU for inspection.

We describe the two components separately in the following sections, using the example shown in Figure 2.

Example: Figure 2 shows a simple C program, which takes two integers as command-line inputs, `a` and `b`, and adds them to a global variable `c`, using the `addc` function. It prints the results to standard output. A test case for this program would consist of values for the two input integers - `a` and `b`, and the expected result would be a value for the variable `sum`.

4.1 CodeGen

The main task of our code generator is to convert the original C program into an OpenCL kernel. As seen in Figure 1, the CodeGen tool takes the `unmodified` C source code of the tested program and a configuration file.

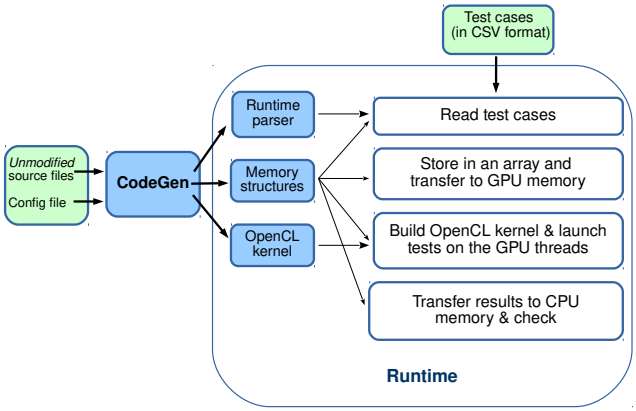

Figure 1: Automated Test Execution on the GPU

4.1.1 Configuration. The configuration file describes the test case inputs and results for the tested program. It is used by CodeGen to generate the data structures which are used to transfer test inputs and results between the CPU and GPU memories. Figure 3 shows the configuration file for the example program and the data structures generated by the tool. The configuration file shows that the program takes two integer inputs through the command line, corresponding to `argv` indices 1 and 2, and produces a single integer result, which corresponds to the `sum` variable. This simple program uses integers, but we also support the use of custom data types, pointers and arrays for the test case inputs and results.

As seen in Figure 3, the configuration is directly translated into data structures for the test case inputs - `struct test_input`, and test case results - `struct test_result`. CodeGen outputs them in a header file `structs.h`, which is used both by the CPU and GPU code.

CodGen also generates a parser for the Runtime, which is used to read values for the test cases and assign them to members of the generated memory structures.

4.1.2 Kernel Generation. Once the main data structures have been created, CodeGen translates the tested program into an OpenCL kernel which executes on the GPU threads. This is achieved using a compiler-based approach which works at the AST (Abstract Syntax Tree) level using the Clang C frontend.

Figure 2 shows the kernel generated for the example program. CodeGen changes the signature to the main function, which now takes two arguments, (1) the test inputs; values for which are initialised by the CPU, and (2) the test results, which will be calculated by the kernel. The memory structures `test_input` and `test_result` are the ones generated by CodeGen, as described in Section 4.1.1. As seen in Figure 2, each GPU thread, identified by its global id (`idx`), selects a different test case for execution (`input_gen`) as well as a different test result, in which to record its outputs (`result_gen`). CodeGen replaces reading of input parameters from `argv` with assignments from test case `input_gen`. It also also adds an assignment of the results of the test, variable `sum`, to `result_gen`.

In this way, CodeGen generates an OpenCL kernel for the tested program. It is important to note that CodeGen does not change the core algorithm of the program, but only its input/output interface, ensuring that the tested functionality is the same.

4.1.3 Code Transformations. While generating the OpenCL kernel, CodeGen performs code transformations for C features, which are not readily supported by the OpenCL standard.

Global scope variables. OpenCL does not support assignment to global scope variables. CodeGen moves them to local scope by moving their declarations into the kernel function, `main_kernel`, and passing them as arguments to any functions using them. By using pointers, the tool ensures that any changes made to their values would be visible to the all the other functions, preserving the "global" nature of the original variable. This is shown in our example in Figure 2 with global variable `c`.

Command line arguments. Values for command line arguments should be supplied as part of the test case input. As seen in Section 4.1.2, CodeGen replaces references to the command line arguments with references to the corresponding values in the generated `test_input` structure.

Standard input and output. Similar to command line arguments, a value for every standard input needs to be supplied as part of the test case. CodeGen then replaces references to the standard input with references to the memory structures containing the test cases. Standard output is commented out by CodeGen, except in the cases when it is specified as the expected testing result in the configuration file. In those cases, the tool replaces it with a write to the generated `test_result` structure.

Standard library calls. The OpenCL standard does not support calls to the C Standard Library. To tackle this problem, a custom OpenCL implementation of the C Standard Library is necessary. For this project, we implemented a small subset of Standard Library functions in OpenCL, namely functions in `ctype.h`, `string.h`, `atoi` and `fgets`. We took inspiration from **uClibc** [4], a very small C standard library typically used for embedded systems. In our future work, we plan to add OpenCL implementations of other standard library functions as the need for them arises.

4.1.4 Implementation details. CodeGen is implemented in C++14, using the Clang LibTooling library [5]. It consists of two main components: *Kernel Generator* and *Data Structure Generator*. The Kernel Generator performs the code transformations which translate the original C program into valid OpenCL kernel code. It uses LibTooling's AST Matchers to perform sequential compiler passes, which find and transform the relevant portions of the original program. It then uses the Rewriter class inside the corresponding AST Handlers to perform the transformations at source code level. The Data

Listing 1: Original C program.

```
#include <stdio.h>
#include <stdlib.h>

int c;

int addc(int a, int b){
  return a + b + c;
}

int main(int argc, char* argv[]){

  int a = atoi(argv[1]);
  int b = atoi(argv[1]);
  c = 3;

  int sum = addc(a, b);

  printf("%d + %d + %c = %d\n", a, b, c, sum);
}
```

Listing 2: Automatically generated OpenCL kernel.

```
#include "structs.h"
//#include <stdio.h>
//#include <stdlib.h>

/*int c;*/
int addc(int a, int b, int *c){
  return a + b + (*c);
}

kernel void main_kernel(
    global struct test_input* inputs,
    global struct test_result* results){

  int idx = get_global_id(0);
  struct test_input input_gen = inputs[idx];
  global struct test_result *result_gen = &results[idx];
  int argc = input_gen.argc;
  result_gen->test_case_num = input_gen.test_case_num;

  int c;

  int a = input_gen.a;
  int b = input_gen.a;
  c = 3;

  int sum = addc(a, b, &c);

  /*printf("%d + %d + %c = %d\n", a, b, c, sum);*/
  result_gen->sum = sum;
}
```

Figure 2: Example of converting a C program into an OpenCL GPU kernel using CodeGen.

Listing 3: Configuration file.

```
input: int a 1
input: int b 2
result: int sum variable: sum
```

Listing 4: Generated data structures - file structs.h.

```
typedef struct test_input{
  int test_case_num;
  int argc;
  int a;
  int b;
} test_input;

typedef struct test_result{
  int test_case_num;
  int sum;
} test_result;
```

Figure 3: Configuration file and generated data structures for the example program.

Structure Generator performs a straightforward translation of the configuration file into data structures, and CPU code to read and write to them. The source code for CodeGen, along with instructions to build and execute it, can be found at https://github.com/issta2017sub/CodeGen.

4.2 Runtime

The Runtime implements the CPU's functionality for launching test cases for execution on the GPU. This functionality is always the same for any tested C program. It consists of reading values for the test cases from a CSV file, building the OpenCL kernel for the tested program, launching the kernel with the supplied test cases and checking that the test results are correct. The Runtime is implemented in standard C, using the OpenCL API to perform all the GPU related operations. The source code, along with instructions to build and execute it, can be found at https://github.com/issta2017sub/Runtime.

Test Cases. Test cases are provided in a CSV (Comma Separated Value) file in which, (1) each row corresponds to a test case, (2) the first column contains the id of the test case, (3) the subsequent columns contain the input variables, in the order in which they are given in the configuration file.

For the example in Figure 2, a test case file with 5 test cases could look like this:

```
1     13      7
2     50     22
3   1000      0
4      0   1000
5      0      0
```

where the values for inputs `a` and `b` for test case 1 are 13 and 7 respectively, for test case 2 they are 50 and 22 and so on. The Runtime also supports custom data structures and arrays for the test case inputs. The values for them are given in the same CSV format. To parse and assign values to the members of `struct test_input` correctly, the Runtime uses code, generated automatically by our CodeGen tool.

Kernel build and launch. After the Runtime has read the values of the test cases, it stores them in an array of type `sruct test_input`. This array is transferred to GPU memory, from where each thread reads its respective test case and executes it, as discussed in Section 4.1.2 and shown in Figure 2. The Runtime builds and launches the OpenCL kernel generated by CodGen, using the standard OpenCL API.

Transfer results and check. Once the GPU kernel has executed, we transfer results back to the host where the results are validated against the gold output. Any difference observed is recorded and presented to the user.

5 EXPERIMENT

We check the feasibility and performance of our approach on C programs from the embedded systems domain. We seek to assess three aspects in our evaluation, overhead of using a GPU, speedup over single and multi-core CPU execution, and correctness.

Kernel Execution versus Data Transfer Time: GPU programs have to copy data back and forth from the host memory to perform I/O or when GPU memory is exhausted. Data transfer between the GPU and host memory is slow due to the high latency of the interface. We measure the time taken to perform the input and output data transfers between host and device and compare it to the computation time on the GPU.

Speedup: For each subject program and associated sets of test cases, we perform test executions on the (1) **CPU** – that runs the original benchmark and tests without any change on (i) 1 core sequentially, (ii) 2 cores, (iii) 4 cores, and (iv) 8 cores. For multi-core execution, we use OpenMP to launch test executions in parallel. (2) **GPU** – by transforming the program interface and generating kernel and host code through our tool. We measure and compare time taken for executions on the CPU and GPU to assess speedup achieved.

Correctness: We assess that our approach and tool preserves correctness by not altering program functionality when running tests on the GPU. For each subject program and test suite, we collect outputs from test executions on the CPU and those from the GPU and check if they are an exact match.

5.1 Measurement

In our experiments, we use the following CPU – Intel(R) Xeon(R) CPU E5-2640 v3 processor with 8 cores at 2.60 GHz and 16 GB RAM. All the programs were compiled with GCC with the highest optimization level (`-O3`). The GPU we use is the NVidia Tesla K40m with 15860 work items, spread across 15 compute units. The GPU operates at 745 MHz and has 12 GB global memory and 50 KB local memory.

To measure GPU kernel execution time and time taken to transfer inputs/outputs to/from the GPU we use the profiling functions contained in the OpenCL API. For CPU execution time, we use the standard C function *gettimeofday*. Each subject program with each test suite (with sizes from 2^8 to 2^{17}) is run 100 times to measure execution time on the CPU and another 100 times with the GPU to measure execution time and data transfer overhead. We report median execution times and transfer times.

5.2 Subject Programs

We use 9 benchmarks from the Embedded Microprocessor Benchmark Consortium (EEMBC), which provides a diverse suite of benchmarks organised into categories that span numerous real-world applications, namely automotive, digital media, networking, office automation and telecom, among others [17]. EEMBC benchmarks are *not* just processor-based. They focus heavily on embedded software running on smartphone, tablets, and other embedded systems. The benchmark programs in our evaluation are from the automotive and telecom category. A description of the subject programs is provided in Table 1. The first four programs are from the automotive domain and the remaining five from the telecom domain.

Table 1: Subject programs used in our experiment

Subject	Description	Input/Test (in bytes)	Work-group size
`a2time01`	Angle to time conversion	2000	128
`puwmod01`	Pulse Width Modulation	4849	32
`rspeed01`	Road speed calculation	2000	512
`tblook01`	Table lookup & interpol.	1856	512
`autcor00`	Cross corr. of signals	1024	128
`conven00`	Convolution Encoder	512	32
`fbital00`	Bit allocation	604	128
`fft00`	Fast Fourier Transform	1024	32
`viterb00`	Viterbi Decoder	688	512

5.3 Experimental Setup

The benchmarks have a limited number of test cases associated with them. In addition to these, we generated 131,072

unique test cases for each program, using a random number generator, in order to enable experimentation with large test suites. We thus have, for each subject program, test suites with sizes ranging from 2^8 (256) to 2^{17} (131,072) tests. Table 1 also shows the input size of each test in bytes, the smallest input size is for `conven` benchmark with just 512 bytes of data per test, whilst the largest input is 4849 bytes for `puwmod` benchmark.

Each of the benchmarks in Table 1, along with each of its associated test suites was run through our tool to produce the equivalent GPU version. Our tool did not make any changes to the code implementing the algorithm. The changes were restricted to the test harness and input-output interface of the program.

The last column in Table 1 shows the work-group size used on the GPU for each benchmarks. This number was determined experimentally by conducting an exploration of work-group sizes between 32 (the warp size) and 1024, following common practice.

6 RESULTS AND ANALYSIS

6.1 Kernel vs Data Transfer Time

This section evaluates the GPU execution time, investigating the execution time of the GPU kernel and data transfer separately.

Kernel Execution Time. For small input sizes, kernel execution time remains roughly constant on the GPU, due to the large number of available threads, as can be seen in Figure 4 (second bar). When test suite size increases, the GPU simply solicits more threads for parallel execution of the additional test cases having little effect on execution time with the small suites. This constant trend continues until a saturation point is reached when the test suite size exceeds the number of available GPU threads. We call this point in our evaluation, *threshold size*. After reaching the threshold size, kernel execution time increases linearly with test suite size. In most cases, the threshold size is approximately reached for 16,384 tests.

Data Transfer Time. The overheads for each subject program, input and output transfer time, across test suite sizes are shown in Figure 4 (first bar). As expected, data transfer time increase linearly with the test suite size.

Data Transfer vs Kernel Time. We now look at comparing the total time for data transfer to and from GPU (for inputs and outputs) versus the kernel execution time. As we saw above, kernel execution time remains roughly the same with increase in test suite size until the threshold size is reached, since up until then there are available threads to accommodate the additional test cases. However, once the threshold size is reached and all the GPU cores are busy, subsequent increases in test suite size will result in proportional increase in execution time. Data transfer time, on the other hand, increases with test suite size from the start.

For some benchmarks, such as `fbital`, kernel execution time dominates and is up to 5× higher than data transfer time. This means that the cost of data transfer is minimal compared to the time it takes to perform the computation. However, for other benchmarks, such as `puwmod`, the data transfer time is larger than kernel execution time, implying that the amount of computation required per test is relatively small. In such instances, using a GPU incurs a large overhead due to the need for transferring data to the external device. This overhead could be mitigated by using an on-chip GPU or using pipelining as explained in the next section.

Summary. Increasing test suite size proportionally increases input and output transfer overheads. There is no general trend with respect to total transfer time versus kernel execution time. Time taken for data transfers can be hidden by splitting the test suite into smaller groups of tests and pipelining their execution.

6.2 Speedup against CPU

We assess speedup achieved when executing test suites on the GPU compared to (1) single thread execution on CPU, and (2) Multi-core CPU, with parallel execution on 2, 4, and 8 cores.

6.2.1 GPU vs Single Thread. For each subject program and each of the associated test suite sizes (ranging from 2^8 to 2^{17}), we collect the following data: 1. *Execution time on the CPU*, 2. *Execution time on the GPU* to execute all the tests in the suite, *including* the time taken to transfer test inputs from host to GPU and test results back to host. We report time on the GPU *without* optimising (by pipelining) for data transfer on large test suites, which in essence is the worst-case time of our approach. We compute speedup as CPU execution time divided by GPU execution time. The speedup for each of the subject programs over different test suite sizes (on log base 2 scale on the x-axis) is plotted in Figure 5.

Figure 5 clearly shows that for all the programs, speedup increases linearly with increase in test suite size until a point, after which it begins to stabilize. The reason for the linear increase in speedup with respect to test suite size is because increase in test suite size (double) causes a proportional increase (double) in execution time on the CPU. However, on the GPU, execution time remains roughly the same with increase in test suite size until threshold size is reached. After reaching threshold size, speedup generally remains stable (although slow increases are observed in some benchmarks) with increasing test suite size. This is because, both CPU and GPU execution times increase at similar rates with test suite sizes after threshold size causing little change in speedup. For our subject programs, we find that the threshold size where speedup stability starts to be reached is at the test suite size of 16384. Note that for subject programs – `fbital, viterb`, significant increases in speedup continues to be observed up to test suite sizes of 131072 (by 20% rather than 100% as observed before threshold size). For these two programs, the GPU execution time increases at a slightly lower rate than CPU execution time.

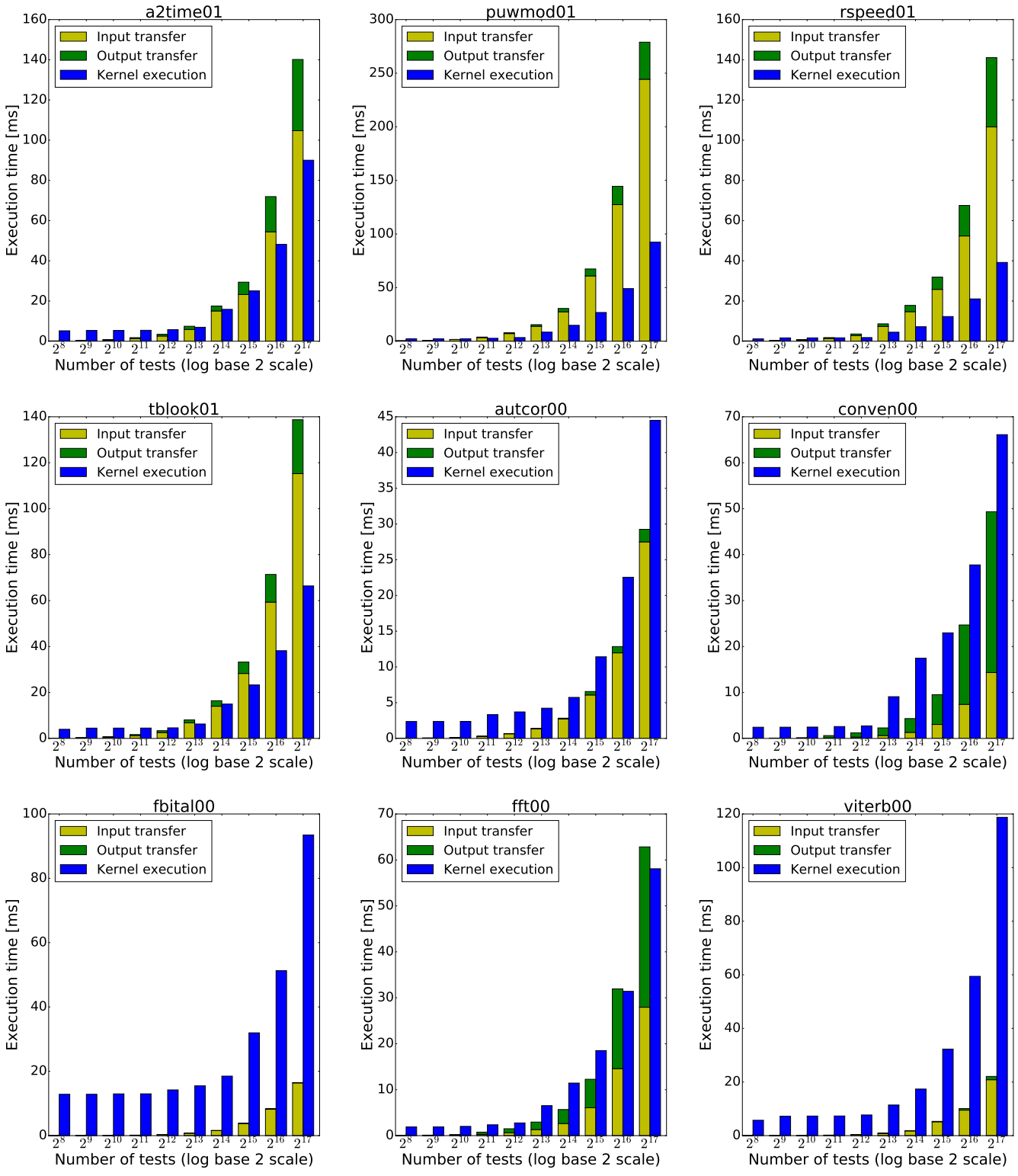

Figure 4: Data transfer overhead versus Kernel Execution time for each subject program

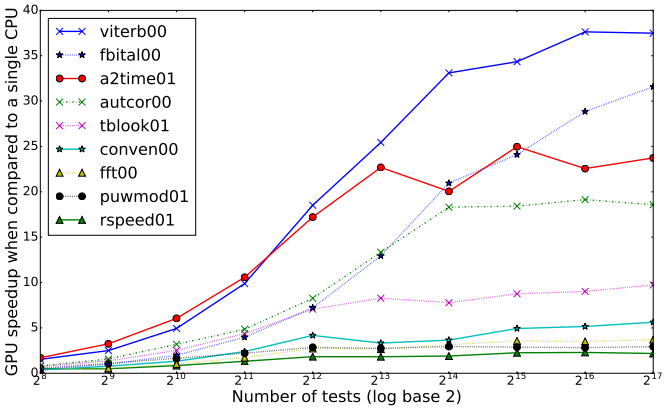

Figure 5: Speedup on the GPU vs single thread execution for 10 different test suite sizes. GPU data transfer and kernel execution not overlapped.

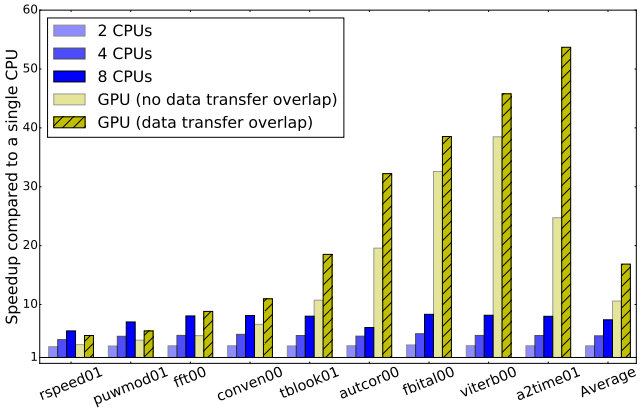

Figure 6: Speedup of GPU and multi-core CPUs over single CPU core.

As seen in the plots in Figure 5, speedup is generally higher when test suite sizes are very large. The average speedup for maximum test suite size in our experiment across all programs is 9.6×. For the smallest test suite size of 256 in our experiment, speedups are limited. We find that running on GPU starts to achieve speedup gains over the CPU when test suite sizes are 2048 or more. This is not surprising, since for smaller test suite sizes, there is not enough workload for the GPU (idle threads) to be better than the CPU. A single execution of the program, in general, is likely to be much faster on the CPU versus the GPU, since the CPU is optimized to run general purpose programs and there is no parallelism within a single test execution that the GPU can leverage.

It is easy to see from Figure 5 that most of the subject programs gain significantly from test execution on the GPU with large test suite sizes. `viterb, fbital, a2time, autcor, tblook` all do well on the GPU with speedups ranging from 9.7× to 37× when executing the largest test suite. Only `rspeed, puwmod, and conven` achieve limited speedup gain 2.2× to 5.6×. Section 6.3 analyzes the reasons for the varying speedup that we observe over the different programs.

6.2.2 Hiding Data Transfer. One of the main issues when offloading computation to a GPU is often the long data transfer time required to send the data to the GPU. This is especially a problem with large test suite sizes. It is possible to mitigate the effect of high transfer time using pipelining to overlap GPU execution and data movement. Large test suites can be split into several smaller groups of tests. While one group of tests executes on the GPU, we can safely start the data transfer for the next group of tests and keep feeding the GPU enough data to process so as to maximize its utilization.

Figure 6, shows the speedup achieved with and without overlapping the data with kernel execution on the GPU. As can be seen, overlapping data transfer always leads to slightly better overall performance on the GPU, bringing the largest speedup from 38× (for `viterb`) to 53× (for `a2time`). For all programs, our approach is able to outperform the CPU execution time (i.e. speedup larger than 1) and offers an average speedup of 16×.

6.2.3 GPU vs Multi-core. To offer a fairer comparison point, we show in Figure 6 speedup achieved when using multiple cores, 2,4 and 8, on the CPU with the largest test suite, by modifying the original application with OpenMP for all subject programs. For most programs, speedup scales linearly with number of CPU cores, achieving an average speedup of 6× with 8 cores over all subject programs. For 5 of the 9 programs, GPU offers significant speedup ranging from 18× to 53×. Average GPU speedup over all benchmarks is 16× as opposed to 6× for an 8-core CPU. This clearly demonstrates the benefit of using a GPU, even when compared to a multicore CPU where each core is fully utilized. In two cases, `rspeed` and `puwmod`, the GPU is slower than using 8 CPU cores. We discuss reasons for the limited speedup in the next Section.

6.3 Analysis

This section presents an analysis to explain the diversity of speedup observed on our benchmarks. It is based on the general observation that the more compute-intensive a program is, the higher the performance will be on the GPU. We use the following metric to establish the computational intensity of a benchmark: $Time\_cpu/Byte$. This represents the time it took for the CPU to process a Byte of input data. A higher value of this metric generally means that more computations are executed per Byte of input data processed, suggesting that the benchmark is more computationally demanding.

Figure 7 shows the computational intensity of each benchmark. As can be seen, the benchmarks with the lowest value (`rspeed`, `puwmod`, `fft`), also have the lowest speedup (Figue 6). Conversely, benchmarks with a high value, `fbital`, `viterb`, `a2time` are the ones exhibiting the largest speedups. `a2time`

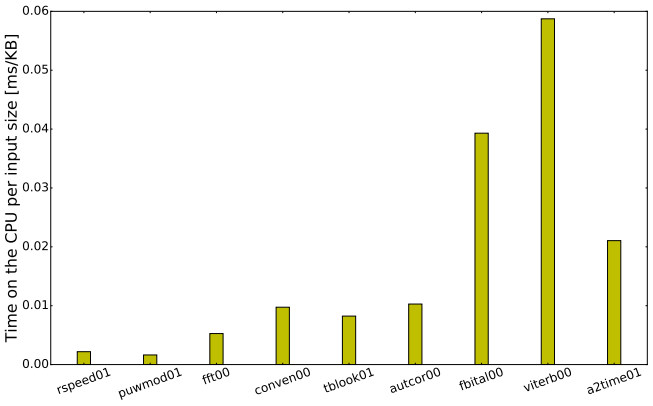

Figure 7: Computational intensity of benchmarks ordered by speedup achieved.

does not have the highest computational intensity but it achieves largest speedup due to the effect of pipelined data transfer. Input data transfer time for `a2time` is comparable to execution time and pipelining groups of tests helps double speedup, approximately, from 24× to 53× .

Interestingly, `conven`, based on its computational intensity, would be expected to perform relatively well, but its speedup is comparable to `fft`, that has lower computational intensity. The reason for this behavior lies in the fact that `conven` uses a temporary array of 1KB for each input test, that is read and written many times in a loop. This array can easily fit in the large cache of the CPU, but cannot fit in the smaller cache of the GPUs, leading to GPU performance being lower than expected.

To summarise, benchmarks that exhibit a high compute-intensity tend to give high speedup when executing test cases on the GPUs. However, performance on the GPU might also be limited by the presence of a data working set larger than the cache, resulting in sub-optimal performance. Given the trend towards larger cache sizes on GPUs, we expect that programs characterised by this feature will exhibit larger speedup on future generations of GPUs.

6.4 Correctness

For each subject program, we collected the test case outputs from the CPU and GPU executions across all test suites. Each test suite was executed 100 times on the GPU and CPU. We found that for all 9 subject programs, with 256 to 131,072 test cases each, the test case outputs between the CPU and GPU executions were an **exact match**. We can safely conclude that our framework for executing tests on the GPU *preserves correctness of program execution* for all 9 embedded system benchmarks and test suites in our experiment.

7 FUTURE WORK

Our approach in this paper handles challenges with respect to usability and performance when running test executions on GPUs. Nevertheless, our tool still lacks support for execution of certain program features on GPUs. It is worth noting that features like dynamic memory and file system calls that GPUs do not support are not commonly present in embedded software. In our future work, we plan to address the following GPU limitations,

System Calls. To address the limitation with system call support, we will provide mechanisms for executing system calls and file I/O on the CPU while concurrently executing the GPU kernels. We will also explore the feasibility of using recently proposed file system abstraction–**GPUfs** [21]– for invoking file system calls efficiently. Additionally, we will investigate profiling techniques to detect system call patterns, similar to the one proposed by Fadel [9], and explore hiding patterns (such as memory management patterns) that do not add valuable information to program behavior.

Control Flow Divergence. To mitigate the impact of control flow divergence, we will analyze and profile tests so that tests with the same or largely similar control flow are grouped together. We will schedule this group of tests onto the same warp or block on the GPU for efficient use of lock-step execution. We believe this technique will be effective since for a large test suite, it is likely there will be substantial numbers of tests with similar control flow.

Recursion Recursion is a feature that is currently not supported in OpenCL. Recursive function calls can, however, be eliminated using a stack implementation in its place. We plan to implement a compiler pass that transforms all recursive calls in the original C/C++ implementations to use a custom stack implementation in OpenCL.

8 CONCLUSION

This paper has shown how test executions of C embedded software can be transparently accelerated using a GPU. We have taken a compiler approach to automatically convert sequential C program into OpenCL kernels. As shown in this paper, this involves the application of compiler transformations for dealing with command line arguments, standard input/output, global variables and library calls. While there is still significant remaining work to support all the features commonly found in C programs, we have demonstrated that this approach works in practice for nine benchmarks from the EEMBC benchmark suite. We achieved significant speedups of up to 53× with the GPU versus 7× with an 8 core CPU.

REFERENCES

[1] http://www.nvidia.com/object/cuda_home_new.html.
[2] https://www.khronos.org/opencl/.
[3] https://www.khronos.org/sycl/.
[4] https://www.uclibc.org/.
[5] http://clang.llvm.org/docs/LibTooling.html.
[6] Samer Al-Kiswany et al. StoreGPU: exploiting graphics processing units to accelerate distributed storage systems. In *HPDC*. ACM, 2008.

[7] Ethel Bardsley et al. Engineering a static verification tool for gpu kernels. In *Proceedings of CAV*, 2014.
[8] Muthu Manikandan Baskaran, J. Ramanujam, and P. Sadayappan. Automatic C-to-CUDA code generation for affine programs. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 6011 LNCS, pages 244–263. Springer Berlin Heidelberg, 2010.
[9] Waseem Fadel. *Techniques for the abstraction of system call traces to facilitate the understanding of the behavioural aspects of the Linux kernel*. PhD thesis, Concordia University, 2010.
[10] Tobias Grosser, Armin Groesslinger, and Christian Lengauer. POLLY - Performing Polyhedral Optimizations on a Low-Level Intermediate Representation. *Parallel Processing Letters*, 22(4):1250010, dec 2012.
[11] Mats Per Erik Heimdahl and George Devaraj. Test-suite reduction for model based tests: Effects on test quality and implications for testing. In *ASE*, 2004.
[12] Laura Inozemtseva and Reid Holmes. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the 36th ICSE*. ACM, 2014.
[13] Colin JW Kushneryk and Paul D Barnett. Parallel test execution, April 26 2012. US Patent App. 12/911,739.
[14] Ding Li, Yuchen Jin, Cagri Sahin, James Clause, and William GJ Halfond. Integrated energy-directed test suite optimization. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 339–350. ACM, 2014.
[15] Zheng Li, Yi Bian, Ruilian Zhao, and Jun Cheng. A fine-grained parallel multi-objective test case prioritization on gpu. In *SSBSE*. 2013.
[16] J.D. Owens et al. A survey of general-purpose computation on graphics hardware. In *Computer graphics forum*, volume 26. Wiley Online Library, 2007.
[17] Jason A. Poovey, Thomas M. Conte, Markus Levy, and Shay Gal-On. A benchmark characterization of the EEMBC benchmark suite. *IEEE Micro*, 29(5), 2009.
[18] Ajitha Rajan, Subodh Sharma, Peter Schrammel, and Daniel Kroening. Accelerated test execution using gpus. In *ACM/IEEE ASE'14*, 2014.
[19] Toomas Remmelg, Thibaut Lutz, Michel Steuwer, and Christophe Dubach. Performance Portable GPU Code Generation for Matrix Multiplication. In *GPGPU: Workshop on General Purpose Processor Using Graphics Processing Units*, pages 22–31, New York, New York, USA, 2016. ACM Press.
[20] Takashi Shimokawabe et al. An 80-fold speedup, 15.0 TFlops full GPU acceleration of non-hydrostatic weather model ASUCA production code. In *SC*. IEEE, 2010.
[21] Mark Silberstein, Bryan Ford, Idit Keidar, and Emmett Witchel. Gpufs: integrating a file system with gpus. In *ACM SIGARCH Computer Architecture News*, 2013.
[22] Michel Steuwer, Christian Fensch, Sam Lindley, and Christophe Dubach. Generating performance portable code using rewrite rules: From high-level functional expressions to high-performance opencl code. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, pages 205–217, New York, NY, USA, 2015. ACM.
[23] Shin Yoo and Mark Harman. Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification & Reliability*, 22(2), 2012.
[24] Shin Yoo, Mark Harman, and Shmuel Ur. Highly scalable multi objective test suite minimisation using graphics cards. In *SSBSE*, 2011.
[25] Zhao Yu, Jae-Han Cho, Byoung-Woo Oh, and Lee-Sub Lee. Parallel algorithm for generation of test recommended path using cuda. *International Journal of Engineering Science & Technology*, 5(2), 2013.